Extracting Vulnerabilities in Industrial Control Systems using a Knowledge-Based System

Laurens Lemaire
KU Leuven, MSEC, iMinds-DistriNet
Department of Computer Science
Gebroeders Desmetstraat 1, 9000 Ghent, Belgium
laurens.lemaire@cs.kuleuven.be

Jan Vossaert
KU Leuven, MSEC, iMinds-DistriNet
Department of Computer Science
Gebroeders Desmetstraat 1, 9000 Ghent, Belgium
jan.vossaert@cs.kuleuven.be

Joachim Jansen
KU Leuven
Department of Computer Science
Celestijnenlaan 200A, 3001 Heverlee, Belgium
joachim.jansen@cs.kuleuven.be

Vincent Naessens
KU Leuven, MSEC, iMinds-DistriNet
Department of Computer Science
Gebroeders Desmetstraat 1, 9000 Ghent, Belgium
vincent.naessens@cs.kuleuven.be

Industrial Control Systems (ICS) are used for monitoring and controlling critical infrastructures such as power stations, waste water treatment facilities, traffic lights, and many more. Lately, these systems have become a popular target for cyber attacks. Both during their design and while operational, security is often an afterthought, leaving them vulnerable to all sorts of attacks.
This paper presents a formal approach for analysing the security of Industrial Control Systems. A knowledge-based system, namely IDP, is used to analyse a model of the control system and extract system vulnerabilities. In this paper we present the input model of the methodology and the inferences and queries that allow the system to extract vulnerabilities. This methodology has been added to an existing framework where the user can model his system in the modeling language SysML. This SysML model then gets parsed into suitable IDP input. A fully working prototype has been developed and the approach has been validated on a real case study.

Industrial Control Systems Security, Critical Infrastructure Protection, Formal Modeling, IDP

1. INTRODUCTION

Industrial control systems used to be isolated, proprietary systems. The only security concern was physical access to the system. With the evolution of IT in these last decades, this is no longer true. ICS now often consist of Commercial Off-The-Shelf (COTS) components, and are connected to a company network and the internet. These changes have made them easier to use, but also more vulnerable to attacks ENISA (2011).

Typical IT solutions are not always applicable to these systems. Their critical nature introduces additional requirements such as high determinism and response times. Reliability of the network is more important than in most IT applications. For these reasons, applying patches to fix vulnerabilities is not always possible, especially if they require a reboot of the system Tom et al. (2008). Patch management is often an important aspect of maintaining ICS.

Due to their critical nature, attacks on these systems could have disastrous consequences. Previous attacks on ICS illustrate this. Famous examples are the Maroochy Shire sewage spill in Australia Abrams and Weiss (2008), and the Stuxnet worm in Iran Matrosov et al. (2011); Falliere et al. (2011). The former caused 800.000 litres of raw sewage to spill into local parks and rivers, the latter was used to sabotage the fuel enrichment plant of Natanz in Iran. Langner (2013).

The Stuxnet worm was discovered in 2010. Industrial control system security has been a popular research topic since. Organisations such as NIST/ISA/ISO have produced security standards and guidelines for adequately protecting these systems Stouffer et al. (2015); ANSI/ISA (2013); ISO/IEC (2008). This work presents a tool that performs a security analysis of an ICS model based on these standards and guidelines. The modeling is done in SysML, while the analysis is done using the Imperative Declarative Programming (IDP) framework.

Contribution. This paper presents a model-based approach for the security analysis of industrial control systems. The control systems are represented as IDP instances. A logic theory inside the IDP framework then extracts vulnerabilities from the system. Vulnerability databases are included in the input model to extract vulnerabilities at the component level. Rules in the logic theory assess what the impact of these component vulnerabilities is at the system level. The methodology gives feedback about the implications related to security in various settings, for instance attackers of different skill levels can be modeled. The security aspects we currently focus on are authorization and authentication. Our system checks if a provided policy specification is respected by the control system. For instance if the policy specifies that a certain user should not be able to modify a parameter in the system, it is checked whether this is actually the case. A prototype of the tool has been created and has been validated on a real case study.

This logic has been added to a framework which allows the user to model the system with SysML, the Systems Modelling Language. The resulting XML file gets parsed to input that is accepted by the IDP framework. Then the security evaluation takes place Lemaire et al. (2014).

Outline. The structure of this paper is as follows. Section 2 gives an overview of related work. Our case study, which will be used throughout the remainder of this paper, is presented in Section 3. Section 4 details the approach and introduces the IDP system. In Section 5 we discuss the input model that will formally represent the industrial control systems in IDP. Section 6 contains the inferences and queries that are used to extract vulnerabilities. The validation on our case study is presented in Section 7. Finally, Section 8 concludes the paper and contains future work.

2. RELATED WORK

In Pai and Bechta Dugan (2002), the authors use UML system models to automatically create dynamic fault trees (DFTs). These DFTs are then analysed to compute the reliability of a system. Our methodology applies a similar approach. We use the Systems Modeling Language (SysML) instead of UML. SysML extends UML with several new or modified diagrams and is more suited for modeling systems or systems-of-systems. Further, our approach does not aim to draw conclusions regarding reliability, but focuses on security instead. Currently there is a lack of attention for system security in model-based system engineering, as discussed in Oates et al. (2013). Our tool fills this gap.

Other tools exist that perform a security analysis on industrial control systems. CySeMoL estimates the probability that attacks succeed against an enterprise system Sommestad et al. (2013, 2010). Similar to our method, CySeMoL allows users to change their system architecture and view the resulting changes on the attack probabilities. For their attack probabilities, CySeMoL assumes that the attacker is a penetration tester who only has access to public tools. More powerful attackers must be considered. Our tool also incorporates vulnerability databases to extract vulnerabilities at the software and hardware level.

The ADVISE security modeling formalism LeMay et al. (2011) establishes which way an attacker is most likely to go about attacking a system. To this end, both the system and the attacker are modeled. ADVISE has been implemented in the Möbius framework Ford et al. (2013) to make use of its modeling formalisms and solution techniques. ADVISE does not draw from vulnerability databases, it assumes the vulnerabilities in the system are already known. Our tool attempts to find such vulnerabilities and could hence be used in combination with ADVISE.

The result of our system analysis is also different from the one in the above tools. ADVISE and CySeMoL calculate the probability that certain attacker goals can be reached. Our approach will check whether a provided policy specification holds true in the industrial control system. The modeler will submit a policy matrix detailing which users can do certain operations on the process parameters. Our logic component will then check whether this specification holds true in the control system. If this is not the case, traces will be returned to detail which vulnerabilities or component properties cause the specification to not hold true.

Tools exist that take vulnerability databases into account, but these focus mainly on network security and are not tailored to ICS. For instance MulVAL Ou et al. (2005) is a tool that models networks in Datalog to perform a network vulnerability analysis. The tool uses an OVAL scanner to find reported software vulnerabilities in a network and returns their impact on the system.

A similar tool is Cauldron Jajodia et al. (2011). Cauldron contains several vulnerability databases, such as NIST's National Vulnerability Database (NVD), the Bugtraq security database, Symantec Deepsight, etc. It integrates with vulnerability scanners such as Nessus, Retina, and Foundscan, to populate its network model. Cauldron draws on these sources to find the vulnerabilities in

a network and automatically generates mitigation recommendations.

Both MulVAL and Cauldron focus on network security. Our tool is aimed at industrial control systems, the vulnerability databases we draw from reflect this. Currently our logic theory only contains the ICS-CERT vulnerability database, which contains vulnerabilities in ICS components (PLCs, industrial operating systems, HMIs, historians, ...). More vulnerability databases will be incorporated later.

3. AN INDUSTRIAL HATCHERY

The approach presented in this paper has been tested on a real case study. The case study is a *hatchery*. The incubators used in this hatchery are manufactured by one of our industry partners.

The hatchery consists of sixteen incubators, twelve *setters* and four *hatchers*. Each incubator can hold up to 115200 eggs. Eggs are initially put in one of the setter incubators, where they are turned hourly. Then they get transferred to the hatcher incubators to hatch. Each incubator consists of various sensors and actuators that are connected to a PLC. At the front of the incubator a touchscreen is used for monitoring and controlling the parameters. Each incubator room has a switch that all PLCs in that room are connected to. This switch is connected to a wireless router which can be used for accessing the incubators with a mobile device, using an app that can take control of the touchscreens. The room switches are all connected to a switch in a centralized location. In this location we also find a server that is used by the manufacturer to connect to the hatchery remotely. There is also an industrial PC that logs all the data and can be used to control all incubators.

The touchscreens and industrial PC utilise role based access control. There are currently four different types of users in the hatchery. The least privileged users can look at the different parameters, but not change them. Additionally, they can reset the incubator alarms or turn off the sound. This is meant for cleaning personnel and technicians. The second lowest level is used by the operators of the hatchery. This usergroup can change all parameters of the incubators, including the temperature settings, humidity, CO2 levels, etc. The local managers make up the third level. They are able to do all the above, as well as change operator passwords, export data regarding login lists and alarms to USB, and so on. The highest level is reserved for the manufacturer of the incubators. When they log on remotely using their password, they also get access to additional information regarding errors and failures, so they can assist when problems occur.

4. APPROACH

In this section the approach is presented. Figure 1 gives an overview of the major components of the modeling approach.

The *System Independent Modeling* part consists of the *Vocabulary* ($\mathcal{V}$) and the *Theory* ($\mathcal{T}$). These remain the same for all systems. The Vocabulary consists of three parts, an input vocabulary to specify the types, predicates and functions that are used in the input model, a reasoning vocabulary that defines the same constructs used in the theory, and finally an output vocabulary to represent conclusions. The theory contains the logic that will be applied to the model to extract the vulnerabilities. There are two sets of rules. One set is run on the input model to expand it. A second set with *queries* then gets applied to the expanded model to draw conclusions.

The *Input Model* ($\mathcal{M}_\mathcal{I}$) consists of an *ICS Model*, a *User Model* ($\mathcal{M}_\mathcal{U}$), a *Policy Specification* ($\mathcal{M}_\mathcal{P}$), and a *Vulnerability Model* ($\mathcal{M}_\mathcal{V}$). The *ICS Model* represents the industrial control system. This is done in three layers. At the top layer we model the process specification, which consists of the process parameters and operations. For instance, each incubator has a temperature parameter which can be modified, and an alarm that can be turned on or off. Then we model the architecture of the control system. This includes all the *components* and the *modules* they consist of. An example of a component is a supervision PC which consists of several software modules that allow users to control or monitor system parameters. Finally, security mechanisms of the control system are modeled. This currently focuses on authorization and authentication of users and components. In the future, this framework can easily be expanded to include additional security properties.

In the *User Model*, different user groups can be defined by the modeler. Each group gets the appropriate credentials associated with them, and other properties that determine how powerful they are. For instance, which components or networks in the system they can access. Attackers are modeled in the same way. They can be given the same credentials and properties as one of the user groups to reason about internal adversaries.

The *Policy Specification* is a list of user permissions. The modeler will indicate which users should be able to do operations on system parameters. This specification will then be checked by the logic rules.

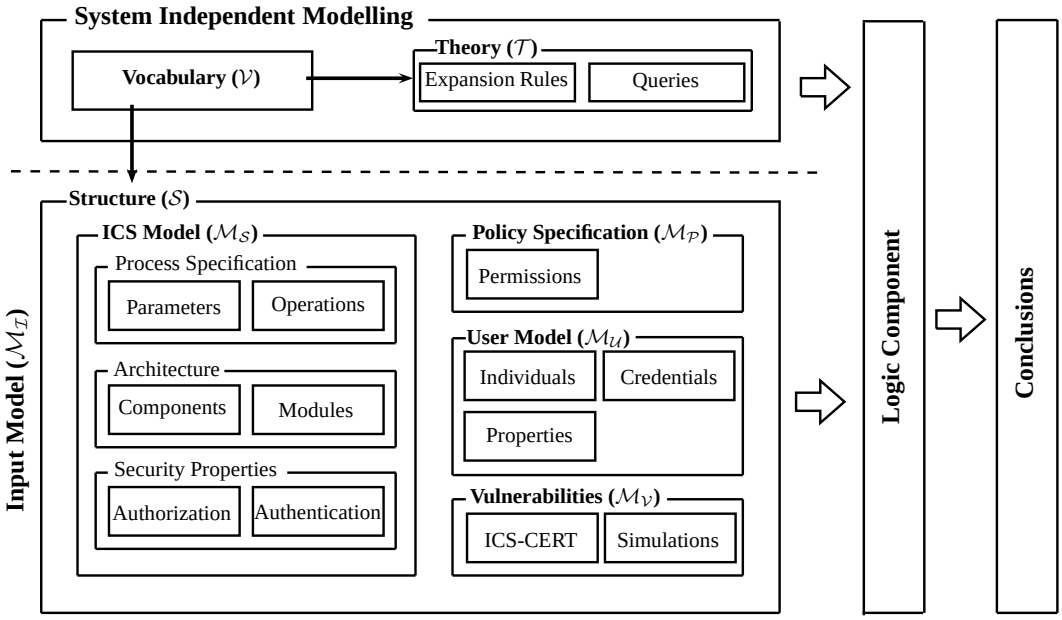

Figure 1: *Framework for extracting vulnerabilities from industrial control systems.*

The *Vulnerability Model* contains vulnerability databases that will identify component vulnerabilities in the system. Currently only the ICS-CSR database is included, but the framework allows for other databases to be added in the future. It is also possible for the modeler to introduce vulnerabilities in components to check their impact on the system security as a whole. For example, a user can mark a certain HMI as being vulnerable to a Denial of Service attack, and then check if the control system can still have its desired functionality, e.g. whether the appropriate user groups can still read and modify parameters they are authorized to.

The *Logic Component* is a logic tool that automates the reasoning on formal models. For our work we have chosen to use the IDP system. IDP is a state-of-the-art declarative programming system developed at KU Leuven. IDP supports reasoning on expressions in a high-level formal language that extends first-order logic, called “The IDP language”. This language adds aggregates, partial functions, inductive definitions, etc. to first order logic to make it easier for users to specify their problem. IDP is used to solve search problems using, amongst other methods, model expansion Wittocx et al. (2008); Bogaerts et al. (2012).

Once a control system is modeled in the IDP framework, the *logic theory* can be used to expand this model, finding a value for previously unknown data that complies with the theory. If at least one such expansion exists, the set of resulting models is returned. If there is no expansion possible from the structure that does not violate the components in the theory, it is possible to identify which theory components were violated by the model.

The major strength of the IDP framework is an intuitive input language, which allows us to model problems fluently. The use of inductive definitions, for example, is very helpful for modeling transitive closures and other recursive predicates.

Once the logic component has finished analysing the model, the *conclusions* are returned. These are the predicates and functions defined in the output vocabulary.

5. SPECIFYING THE INPUT MODEL

In this section we show which information the modeler has to input in order to analyse a particular system. We use the hatchery from Section 3 as an example. Not all predicates and functions are shown here due to space limitations.

5.1. The ICS Model

The *ICS Model* specifies the control system. This is split up in three parts. In the *Process Specification*, the modeler lists out the *Parameters* and *Operations*. *Parameters* are the variables of the environment that the control system interacts with. For instance, the temperature or humidity inside an incubator, the status of incubator lights/alarms, etc. The modeler is free to use his own naming conventions for these parameters. Here we will refer to the temperature of the first setter incubator as parameter $Temp_{S_1}$, and so on. *Operations* are the actions that users of

the system can perform on these parameters. We consider two operations: *Read* and *Modify*.

$type\ Parameter$
$Parameter = \{Temp_{S_1}, Alarm_{S_1}, Temp_{S_2}, \ldots\}$

$type\ Operation$
$Operation = \{Read, Modify\}$

In the *Architecture*, the physical parts of the control system are modeled. A type *SystemPart* is created which will contain all the building blocks of the system. Type *Component* is a subtype of *SystemPart*. *Components* are the basic elements of the systems, for instance a PLC, a touchscreen, a supervision PC, ...

$type\ SystemPart$
$type\ Component\ isa\ SystemPart$
$Component = \{PLC_{S_1}, Switch_{Hatchery}, \ldots\}$

Components can consist of several software processes, which are called *modules* in our model. Consider the industrial PC that is used for monitoring and controlling the incubators at the hatchery. In this PC we will distinguish two modules, one to represent the operating system (OS), and one to represent the software process that can influence the parameters of the control system.

$type\ Module\ isa\ SystemPart$
$Module = \{Control_{IndPC}, OS_{IndPC}, \ldots\}$

Modules are associated with components using a *SoftwareModule* predicate:

$SoftwareModule(Module, Component)$

Components have their product and version information associated with them using a predicate. This will be used to find component vulnerabilities, which is explained in the next section.

$type\ Product$
$type\ Version$
$Switch(SystemPart, Product, Version)$

Using predicates *Measure* and *Control*, components and modules are connected to the parameters they measure or control.

$Measure(Component, Parameter)$
$Measure = \{(AlarmSensor_{S_1}, Alarm_{S_1}), \ldots\}$

$Control(Module, Parameter)$
$Control = \{(Control_{IndPC}, Temp_{S_1}), \ldots\}$

The modeler also indicates which *Networks* are considered in the control system and what components are placed in them. In this context a *Network* is any collection of components that are connected to each other and can exchange information.

$Network = \{Network_{Hatchery}, \ldots\}$
$NetworkLocation = \{(PLC_{S_1},$
$Network_{Hatchery}), \ldots\}$

Finally, the relevant *Security Properties* are associated with the components and modules by using predicates. Currently the focus lies on *Authentication*, *Authorization*, and *Vulnerabilities* introduced by the modeler, but the framework allows for adding more categories of properties.

Authentication is achieved by using *tokens* (passwords, keys, ...). Modules get tokens associated with them. If users possess the correct tokens, they can authenticate themselves to these modules and make use of their functionality.

$type\ Token$
$Token = \{Password_{IndPC}, \ldots\}$
$Authentication(SystemPart, Token)$

Authorization specifies which tokens are required on a certain module to control process parameters. For instance, a certain password $Pw_{Technician}$ might give the user access to the alarm parameters, but not the temperature, whereas password $Pw_{Manager}$ gives access to all.

$Authorization(Pw_{Manager}, Control_{IndPC}, Temp_{S_1})$

If a manager now successfully authenticates to the module using $Pw_{Manager}$, he will be able to control the parameter. Which operations the manager can do on this parameter is modeled in the *policy specification* explained below.

5.2. The User Model

The *User Model* lists the users in the system and assigns properties to them. Various *Attackers* of different strength can be included in the model. They are modelled in the same way as other users and only differ in name.

$type\ User$
$User = \{Technician, Operator, Manager,$
$InternalAttacker, \ldots\}$

Credentials and properties can then be associated with the different user groups or attackers by using the predicates provided in the vocabulary. For instance if we want to specify that the manager has the password required to access the industrial PC, the following lines are added to the model:

$HasToken(User, Token)$
$HasToken = \{(Manager, Password_{IndPC}), \ldots\}$

	$Temp_{S_1}$	$Alarm_{S_1}$	$Humidity_{S_2}$
$Technician$	R	$R\ M$	R
$Operator$	$R\ M$	$R\ M$	$R\ M$
$Manager$	$R\ M$	$R\ M$	$R\ M$
$Attacker$			

Table 1: *Part of the Policy Specification.*

Similarly, other predicates allow the modeler to specify which networks a user has access to, or what components he can physically access. All these predicates can also be assigned to the attackers.

5.3. The Policy Specification

The *Policy Specification* details the permissions of the users. By this we mean which operations the users should be able to do on the control system parameters. The modeler provides this list when modeling the system and then the logic theory will check whether the control system complies to this specification.

$$Permission(User, Parameter, Operation)$$

Table 1 shows a list of permissions from the hatchery case study. This is not the full policy specification, the full example contains too many parameters to list here.

An attacker should not be able to read or modify any parameters in the system. Technicians are able to turn the incubator alarms on or off, but they are not authorized to make any other modifications. The operators and managers have full control over all parameters.

5.4. The Vulnerability Model

The *Vulnerability Model* consists of two parts. One part contains *Vulnerability Databases* that will find *Component Vulnerabilities*. The other part are vulnerabilities introduced by the modeler, allowing the modeler to simulate attacks on the system.

Component vulnerabilities are associated with the software or the hardware of a certain component. For example, a certain version of a PLC might have an error in it that causes a denial of service attack. If this PLC is present in the control system that is being modeled, the user should be made aware of this vulnerability.

To make this possible, the vulnerabilities in the ICS-CERT database have been converted into IDP rules. The database contains alerts and advisories. Alerts inform the user about newly discovered vulnerabilities in ICS equipment, whereas advisories contain fixes and mitigations. Both have been included in IDP Lemaire et al. (2014). Additional vulnerability databases will be included in the logic theory in the future.

Component vulnerabilities are represented with the following predicate:

$$HasVulnerability(SystemPart, Vulnerability)$$

Simulation. We want the user to have the possibility to flag components with a certain type of vulnerability. For instance, if a user wants to find out what would happen if a supervision PC was vulnerable to a denial of service attack, he should be able to flag this PC as vulnerable. This is done by adding a simple predicate:

$$SimulateDoS(SystemPart)$$

There are several categories of component vulnerabilities, and they each have their *Simulate* predicate. This is explained further in the next section.

6. FINDING VULNERABILITIES IN THE ICS

In this section we will focus on the rules in the *Theory* that extract vulnerabilities. We show the interaction between component and system vulnerabilities.

Component Vulnerabilities. The component vulnerabilities are grouped in categories which will be used later. For instance, *BufferOverflow, HeapOverflow, MemoryCorruption, ...* are all vulnerabilities that can potentially cause a *Denial of Service*. Hence we can group them together as follows:

$$\forall x[SystemPart] : HasDoSVuln(x) \leftarrow HasVulnerability(x, BufferOverflow) \vee HasVulnerability(x, HeapOverflow) \vee \ldots$$

All component vulnerabilities are assigned to one or more categories. Other categories include *Escalation of Privilege, Data Leakage, Bypass of Authentication, ...* The categories are based on the threats we currently consider in the system vulnerability rules.

Remember the user is also able to directly flag components as vulnerable. Both ways of finding vulnerabilities are combined with one final disjunction:

$$\forall x[SystemPart] : DoS(x) \leftarrow HasDoSVuln(x) \vee SimulateDoS(x)$$

Now these various categories can be used in system vulnerabilities.

System Vulnerabilities. The vulnerabilities in the previous section are at the component level.

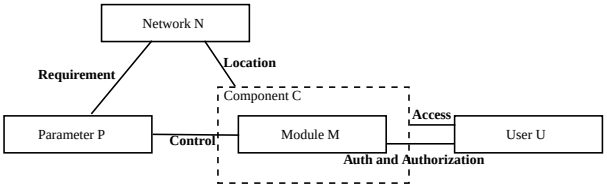

Figure 2: *A conceptual representation of parameter accessibility.*

Having an overview of which components contain vulnerabilities is useful, but system security as a whole is even more important. A set of rules is present in the theory which evaluates this, we refer to these as system rules.

Currently the system rules focus on authentication and authorization. To give the reader a feel for these rules, the rest of this section will give an example of a set of system rules related to authorization that deal with *parameter accessibility*. By this we mean the ability for users to read or modify process parameters, for example reading the temperature of an incubator, or turning of an incubator alarm. The desirable situation is that users are able to control the parameters that they are authorized to control, which is specified by the modeler in the *policy specification*.

Figure 2 shows the parts of the system that are considered in this parameter accessibility example.

If a certain software module is able to control a parameter, a credential needs to be authorized to use the software to change said parameter, the user must also be able to authenticate to the module, and the user must have access to the component that contains the module. The component containing the module must also be able to reach the network of the actuator that controls the parameter in order to send commands to it.

Some of this information is taken directly from the model and represented as predicates. These are listed below:

- **Control(Module, Parameter).** This predicate associates software modules with the parameters they can control. For instance, the control module inside the Industrial PC in the hatchery is able to change all incubator parameters, this is represented in the structure as $Control = \{(Control_{IndPC}, Temp_{S_1}), \ldots\}$.
- **LocationRequirement(Parameter, Network).** A predicate that indicates what network components have to reach in order to control a given parameter.
- **NetworkLocation(SystemPart, Network).** Associates system parts with the network they are in.
- **Authorization(Token, Module, Parameter).** This predicate lists tokens that are required to control parameters on a module. $Authorization(Pw_{Manager}, Control_{IndPC}, S1T)$ indicates that when logging in using the manager password on the industrial PC, it is possible to control the temperature of setter 1.
- **Authentication(SystemPart, Token).** These pairs indicate the authentication details of the components.

The other predicates are initially empty, and get filled up based on the information in the model. Definitions are provided in the logic theory for filling up these predicates. The end result is a predicate *ModifyParameter(User, SystemPart, Parameter)* which includes all triples of users that can control parameters through a certain system part. The definition of this predicate is provided in IDP Listing 1. The different parts are explained below.

The first three predicates on the right side of the definition, *Control(x,z)*, *Authorization(t,x,z)*, and *HasToken(u,t)* are simple checks on the model to see if a user is authorized to use a certain software module that is able to control the parameter.

Next is authentication, the definition says that either the user can prove all credentials of the Module (*AllCredsProven(x,u)*), or there are no credentials required (*NoCreds(x)*). *AllCredsProven(Module,User)* is defined as follows:

$$\forall x[Module]\ u[User] : AllCredsProven(x,u) \leftarrow (\forall\ o[Token] : (Authentication(x,o) \implies HasToken(u,o))).$$

In order to authenticate to a module, the system checks which tokens are required, and then confirms the user possesses all these tokens.

Reachable(User,Module) indicates whether a user has access to a module. This can be either physical access to the module or logical access from a remote component.

$(SoftwareModule(x,y) \wedge LocationRequirement(z,n)) \implies NetworkLocation(y,n)$ says that if module *x* is on component *y*, and the parameter can only be controlled from network *n*, then component *y* must be in network *n* to modify the parameter using module *x*.

$$\left\{ \begin{array}{l} \forall x[Module], y[Component], z[Parameter], u[User], n[Network]t[Token] : ModifyParameter(u,x,z) \leftarrow \\ Control(x,z) \wedge Authorization(t,x,z) \wedge HasToken(u,t) \wedge (AllCredsProven(x,u) \vee (NoCreds(x) \wedge \\ Reachable(u,x))) \wedge ((SoftwareModule(x,y) \wedge LocationRequirement(z,n)) \Rightarrow NetworkLocation(y,n)) \\ \wedge \neg(DoS(x)). \end{array} \right\}$$

IDP Listing 1: The definition for parameter accessibility.

Finally, if there are component vulnerabilities that allow an attacker to take down a module by performing a Denial of Service (DoS) attack, it is possible that the module can not be used, and hence the user can not reach the parameter through the module. This is represented by *DoS(x)*.

Queries. Once the definitions have completed all the predicates and the model is fully expanded, it is possible to check whether certain properties hold. A second theory contains logic rules that do exactly this. These rules are referred to as *queries*.

For the above example, we could check whether the triples in *ModifyParameter* are consistent with the policy specification submitted by the modeler. This would indicate that no unauthorized users are able to change parameters, and the authorized users can. Such a query is constructed as follows:

$$\forall u[User] \quad s[SystemPart] \quad p[Parameter] : \quad ModifyParameter(u,s,p) \quad \Rightarrow Permission(u,p,"Modify").$$

Other queries check various other security properties of the model, e.g. whether users have access to networks they should not have access to, or physical access to protected components; whether users can change device configurations they are not authorized to, etc.

When IDP evaluates and there are no models that satisfy the queries, it can be asked to print a minimal subset of the given theory that is unsatisfiable given the structure. It is shown step by step how a rule in the theory fails. This allows an operator to identify which vulnerabilities are critical when it comes to system security. The printing is achieved by adding the *printcore(theory,structure)* command in the main call.

7. EVALUATION

The case study from Section 3 has been modeled and analysed in IDP. Several vulnerabilities were found.

Our modeling approach works in two iterations. During the first iteration, we use our logic rules to complete all the predicates in the model. This includes the *HasVulnerability* predicate, hence we already identify component vulnerabilities during this iteration. Then, during a second iteration, the queries are run on this completed model, to find security vulnerabilities at the system level.

No component vulnerabilities were found. None of the components used in this hatchery are present in the vulnerability databases that are checked.

At the system level, an authorization vulnerability was found. The HMI touchscreens on the incubators have role-based access control implemented. However, the PLCs they are connected to do not. If an attacker finds a way to overwrite flags in the PLC directly, he can change the parameters of the incubators without being authorized to do so. The PLC programming software provides such a way. The PLC manufacturer provides a free trial download of the programming software on their website. Some basic registration is required, but anyone can access the software. It is then sufficient to get access to the network the target PLC is part of, and know the PLC IP address. Various network scanners can be used to find out the IP address. Once known, the programming software can be used to change this PLC directly.

The way our tool inferred this vulnerability is illustrated in the following listings. The query that is violated is shown in IDP Listing 2, which states that users should only be able to change a configuration of a module, if they have the authorization to use that module for modifying parameters. When the query fails, the trace in IDP Listing is returned, allowing the user to infer why the query has failed. In this case, we see that the configuration of setter 1 can be changed by technicians, allowing them to change parameters of this setter, which they are not authorized to do. The system has inferred the fact that technicians can change this configuration by noting that they have network access to the hatchery network, and access to the PLC programming software.

The full evaluation process takes 2.24 seconds. The runtime scales well with more rules or bigger systems, always finishing in less than 3 seconds.

$$\left\{ \forall u, p, c : (ChangeConfig(u, c) \wedge ConfigAffects(c, p)) \Rightarrow Permission(u, p, "Modify"). \right\}$$

IDP Listing 2: The IDP query that was not satisfied

```
>>> Generating an unsatisfiable subset of the given theory.
>>> Unsatisfiable subset found, trying to reduce its size (might take some time,
can be interrupted with ctrl-c.
The following is an unsatisfiable subset, given that functions can map to at most
one element (and exactly one if not partial) and the interpretation of types and
symbols in the structure:
(~(ChangeConfig("Technician","ConfigurationS1") &
ConfigAffects("ConfigurationS1","TempS1")) | (
Permission("Technician","TempS1","Modify"))) instantiated from line 360
with c="ConfigurationS1", p="TempS1", u="Technician".
Elapsed time to find models : 2.24 sec
```

IDP Listing 3: The final lines of the output, showing the trace of the failed model

This approach has been added to the framework described in Lemaire et al. (2014). The framework allows the user to model the control system in SysML rather than straight in IDP, which can be useful for complex systems. A parser has been written that is able to convert a SysML model XML file into the correct input for the IDP system.

8. CONCLUSIONS

This paper presents a tool that automates the security analysis of industrial control systems. The approach uses modeling and formal reasoning to accomplish its goal. A logic theory in a knowledge-based system extracts vulnerabilities on a component and system level. The rules in the logic theory are taken from vulnerability databases and ICS security standards and guidelines. Once the vulnerabilities are extracted, the user can change the model accordingly to reason about the effects of component changes or newly introduced vulnerabilities on system security.

A first version of the tool is ready and has been tested on a real case study.

Currently the results are being displayed in a text file as standard IDP output. Future work will include representing the vulnerabilities in a SysML diagram that is part of the model. There are also additional sets of system rules that still need to be included in the logic theory.

REFERENCES

Abrams, M. and J. Weiss (2008). Malicious control system cyber security attack case study–maroochy water services, australia.

ANSI/ISA (2013). Ansi/isa-62443-3-3 (99.03.03)-2013 security for industrial automation and control systems part 3-3: System security requirements and security levels.

Bogaerts, B., B. De Cat, S. De Pooter, and M. Denecker (2012). The idp framework reference manual.

ENISA (2011). Protecting industrial control systems: Recommendations for europe and member states.

Falliere, N., L. Murchu, and E. Chien (2011). W32.Stuxnet Dossier.

Ford, M. D., K. Keefe, E. LeMay, W. H. Sanders, and C. Muehrcke (2013). Implementing the advise security modeling formalism in möbius. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pp. 1–8. IEEE.

ISO/IEC (2008). Iso/iec 21827 information technology – security techniques – systems security engineering – capability maturity model (sse-cmm).

Jajodia, S., S. Noel, P. Kalapa, M. Albanese, and J. Williams (2011). Cauldron mission-centric cyber situational awareness with defense in depth. In *Military Communciations Conference, 2011-MILCOM 2011*, pp. 1339–1344. IEEE.

Langner, R. (2013). To kill a centrifuge: A technical analysis of what stuxnets creators tried to achieve.

Lemaire, L., J. Lapon, B. De Decker, and V. Naessens (2014). A sysml extension for security analysis of industrial control systems. In *Proceedings of the 2nd International Symposium for ICS & SCADA Cyber Security Research*, pp. 1.

LeMay, E., M. D. Ford, K. Keefe, W. H. Sanders, and C. Muehrcke (2011). Model-based security metrics using adversary view security evaluation (advise). In *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*, pp. 191–200. IEEE.

Matrosov, A., S. V. Researcher, E. Rodionov, R. Analyst, and D. Harley (2011). Stuxnet Under the Microscope.

NIST (2012). Guide for conducting risk assessments. *NIST special publication 800-30 Revision 1*.

Oates, R., F. Thom, and G. Herries (2013). Security-aware, model-based systems engineering with sysml. In *Proceedings of the 1st International Symposium for ICS & SCADA Cyber Security Research*, pp. 78.

Ou, X., S. Govindavajhala, and A. W. Appel (2005). Mulval: A logic-based network security analyzer. In *USENIX security*.

Pai, G. J. and J. Bechta Dugan (2002). Automatic synthesis of dynamic fault trees from uml system models. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pp. 243–254. IEEE.

Sommestad, T., M. Ekstedt, and H. Holm (2013). The cyber security modeling language: A tool for assessing the vulnerability of enterprise system architectures. *Systems Journal, IEEE 7*(3), 363–373.

Sommestad, T., M. Ekstedt, and L. Nordström (2010). A case study applying the cyber security modeling language.

Stouffer, K., S. Lightman, V. Pillitteri, M. Abrams, and A. Hahn (2015). Guide to industrial control systems (ics) security.

Tom, S., D. Christiansen, and D. Berrett (2008). Recommended practice for patch management of control systems.

Wittocx, J., M. Mariën, and M. Denecker (2008). The idp system: a model expansion system for an extension of classical logic. In *Proceedings of the 2nd Workshop on Logic and Search*, pp. 153–165.